Science of Computer Programming 77 (2012) 424–443

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Mechanised wire-wise verification of Handel-C synthesis

Juan Perna *, Jim Woodcock

Computer Science Department, The University of York, York YO10 5DD, UK

ARTICLE INFO

Article history:
Received 30 January 2009
Received in revised form 3 February 2010
Accepted 9 February 2010
Available online 22 February 2010

Keywords:
Handel-C synthesis
Denotational semantics
Correctness
Mechanical verification
HOL

ABSTRACT

The compilation of Handel-C programs into net-list descriptions of hardware components has been extensively used in commercial tools but never formally verified. In this paper, we first introduce an extension of the compilation schema that allows the synthesis of the prioritised choice construct. Then we present a variation of the existing semantic model for Handel-C compilation that is amenable to mechanical proof and detailed enough for analysing properties of the hardware generated. We use this model to prove the correctness of the wiring schema used to interconnect the components at the hardware level and propagate control signals among them. Finally, we present the most interesting aspects of the mechanisation of the model and the correctness proofs in the HOL theorem prover.

1. Introduction

Handel-C [14] is a Hardware Description Language (HDL) based on the syntax of the C language extended with constructs for dealing with CSP-based [10] parallel behaviour and process communications.

Handel-C's compilation into hardware components (synthesis) was initially formulated to be used in a hardware/software co-design project [20] and later adopted in commercial tools [15]. The synthesis process is described graphically and it covers most of the basic constructs excluding the prioritised choice operator. The approach is based on the fact that each synthesised circuit is controlled by a pair of signals (*start* and *finish*). Compositionality, even though informally addressed, is achieved by means of associating a circuit with each construct and then considering it as a black box driven only by means of the above described simple controlling interface. The correctness of the compilation approach is based on the assumption that the *start* signal will be given to a circuit only if all the previous circuits have already finished. The translation mechanism is designed in a way that, in principle, ensures the validity of this assumption, leading to the satisfaction of a system-level assumption: "the environment will start the hardware program running just once and will not attempt to start it again before the program has completed" [20]. There is, however, no formal evidence of any of these claims as no semantics or formal proofs are provided.

A formal model of the compilation and denotational semantics for the hardware generated has been proposed as a first step towards verifying the compilation scheme [21,24]. The semantics relies on the concept of state-transformers and uses branching sequences of atomic hardware actions as its semantic domain. However, when trying to prove a correctness result from the semantics, we observed that the semantics fails to capture the hardware parallelism at the branching sequence level, making any correctness proof very difficult.

One of the objectives of this work is then to provide a semantics that is capable of capturing the parallelism the hardware is capable of and that is suitable for mechanically verifying the assumptions the synthesis process is based on. As we are interested in a bigger subset of Handel-C than the one addressed in [20], we also need to define a compilation approach for

* Corresponding author.
E-mail address: jiperna@cs.york.ac.uk (J. Perna).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.02.007

⟨program⟩ ::= **main(){** ⟨statements⟩ **}**
⟨statements⟩ ::= ⟨statement⟩
| ⟨statement⟩ ; ⟨statements⟩ | ⟨statements⟩ $\|\|_{HC}$ ⟨statements⟩
⟨statement⟩ ::= **if** ⟨boolean expression⟩ **then** ⟨statements⟩ **else** ⟨statements⟩
| **while** ⟨boolean expression⟩ **do** ⟨statements⟩
| ⟨variable list⟩ $:=_{HC}$ ⟨expression list⟩ | **delay**
| ⟨channel name⟩**?**⟨expression⟩ | ⟨channel name⟩**!**⟨expression⟩
| **priAlt {**⟨guards⟩**}**
⟨guards⟩ ::= ⟨guard⟩ ; ⟨guards⟩ | ⟨guard⟩ | **default:** ⟨statements⟩
⟨guard⟩ ::= **case** ⟨channel name⟩**?**⟨variable name⟩**:** ⟨statements⟩ ; **break**
| **case** ⟨channel name⟩**!**⟨expression⟩**:** ⟨statements⟩ ; **break**

Fig. 1. Restricted syntax for Handel-C programs.

the `priAlt` construct (prioritised choice) as well as providing the semantics corresponding to it. Finally, given the fact that we focus on the correctness of the synthesis process, we exclude any efficiency/optimisation analysis from this work.

The work presented in this paper is a first-order logic theory that describes the syntax (by means of a deep embedding of the constructs as datatypes) and semantics for our subset of Handel-C. The syntax and semantics are then embedded in higher order logics to allow the verification of correctness properties associated with the assumptions the synthesis process is based on. The theory and its verification have been mechanised in the HOL theorem prover [9] and further details about its definitions and theorems can be found in [22].[1] In this paper we intend to give a self-contained and accessible account of that work. In doing so, we deliberately omit all proofs and consequently many technical details in favour of a clearer presentation of the work as a whole.

The rest of the paper is organised as follows. We start by outlining the compilation approach presented in [20] together with our approach for compiling the `priAlt` construct. We then propose a reformulation of the semantic domain in [21] in terms of a slightly higher level semantic domain that allows us to capture parallelism at the sequence level. We then redefine the semantics in terms of the new domain and prove the existence of fix-point solutions for the recursive equations. Finally, we define the concept of wire-satisfiability in our model and prove the correctness of the wiring schema used in the compilation.

2. Handel-C and its synthesis

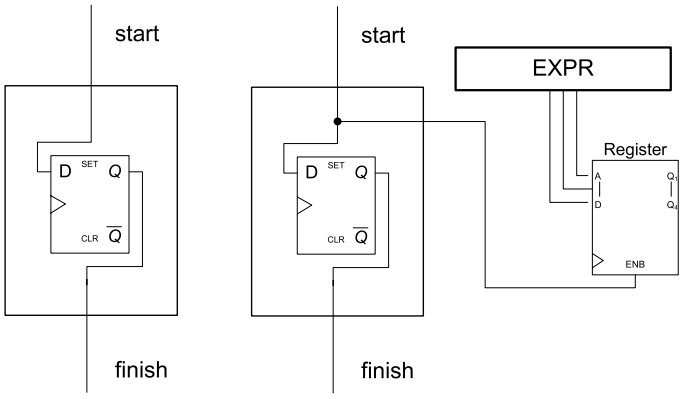
In order to explain how the synthesis process is performed and to provide semantics for the language, a simplified subset that captures the major constructs in the Handel-C language is being used. Most constructs in the language can be built by combining constructs in this subset, with the exception of pointers and function calls. Our subset of Handel-C constructs is presented in Fig. 1.

As described in the language documentation [14], programs are comprised of at least one **main** function and, possibly, some additional functions. Multiple main functions (within the same file) produce the parallel execution of their bodies under the same clock domain. In the context of our elementary subset of the language, it is possible to produce the same effect by means of the parallel operator.

All C-based constructs in Handel-C behave as defined in ANSI-C [12] but with some additional restrictions regarding the clock-based, synchronous nature of the language. In this sense, the evaluation of expressions is performed by means of combinatorial circuitry and it is completed within the clock cycle in which it is initiated (expressions are said to be evaluated "for free" [14] due to this interpretation). This way of evaluating conditions affects the timing of all the constructs in the language. In the case of selection, the branch selected for execution (depending on the condition) will start execution within the same clock cycle in which the whole construct is initiated. The `while` construct starts its body in the same clock cycle in which its condition evaluates to the logical value true. On the other hand, it terminates within the same clock cycle in which its condition becomes false. The assignment construct updates the value of the variable being modified at the end of the clock cycle. This definition of the assignment construct allows swapping the contents of a pair of variables without the need to allocate any temporary memory. Variable declarations are assumed to be implicit; thus, no variable declaration constructs are included in our subset of Handel-C.

Parallel composition of statements executes in a *real* parallel fashion as it refers to independent pieces of hardware running in the same clock domain. The `delay` construct leaves the state unchanged but takes a whole clock cycle to finish. The input and output constructs have the standard blocking semantics: if the two parts are ready to communicate, the value output at one end is assigned to the variable associated with the input side. Both sides of the communication take one full clock cycle to successfully communicate. A process trying to communicate over a channel without the other side being ready will block (delay) for a single clock cycle and try again.

[1] The HOL definitions and proof scripts can be found as an electronic appendix to this paper or downloaded from http://www.cs.york.ac.uk/~jim/WiringVerification/wireVerification.thy.



(a) Delay. (b) Assignment.

Fig. 2. Synthesis schema for the primitive constructs.

Finally, the `priAlt` construct provides a list of possible actions guarded by communication primitives. The alternatives are offered to the environment in descending order (i.e., there is a decreasing priority ordering along the list of guards). Two aspects are worth noticing as regards the `priAlt` construct:

1. The conditions guarding each alternative do not behave like the communication primitives described above (even thought their syntax is the same). As mentioned before, input/output constructs will iterate waiting until the communication is possible. The guards inside a `priAlt` will attempt the communication. If they succeed, the communication will take place and the actions associated with them will start executing in the next clock cycle. If the communication is not possible, the control will be passed to the next alternative in the list of guarded commands inside the `priAlt` (i.e., no iteration will take place inside the guard).
2. The presence/absence of a default clause radically changes the behaviour of the `priAlt` construct. If there is a default clause, the actions associated with it will be executed if all the other alternatives have failed. If the default actions are triggered, control will be passed to them within the same clock cycle in which the `priAlt` construct has been started and the `priAlt` will finish its execution after (but within the same clock cycle) the default actions have finish.

   If there is no default clause and all the alternatives have failed the `priAlt` construct will do nothing until the end of the clock cycle and will offer all its alternatives again (following the priority order) in the next clock cycle.

2.1. The synthesis process

In this subsection we recast the compilation templates presented in [20] that we will formalise and verify in the following sections.

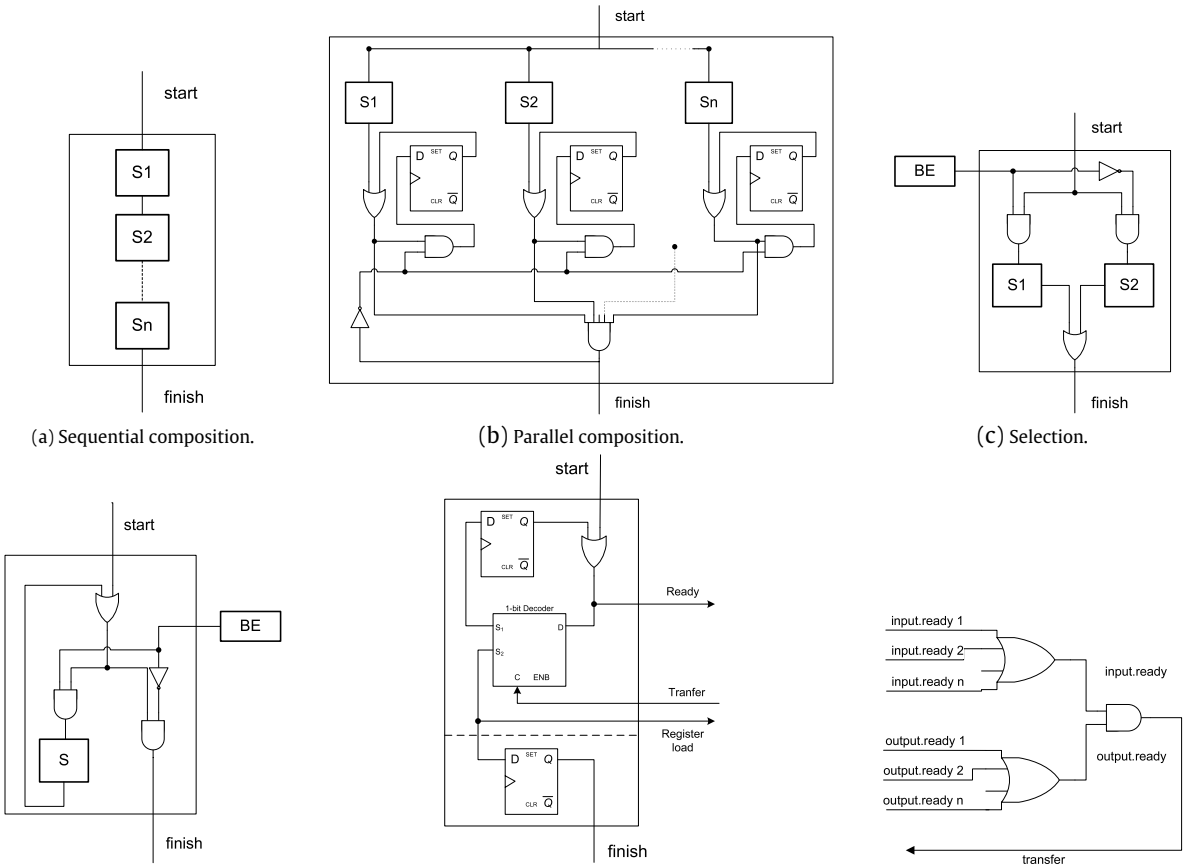
The circuit representing the delay construct (Fig. 2a) takes advantage of the flip-flop's latency to propagate the start signal into its finish wire one clock cycle after receiving it. On the other hand, the synthesised version of the assignment construct (Fig. 2b) uses the same approach the delay circuit uses to preserve the timing model but also uses the start signal to enable the register storing the contents of the variable being assigned to.

Sequential composition of constructs is achieved by means of linking the finish wire with the start one of the corresponding circuits following the sequential ordering (Fig. 3a). The parallel composition of constructs, on the other hand, propagates its start signal to all the parallel processes to achieve their simultaneous activation. Given the fact that each of the parallel branches may take a different amount of clock cycles to terminate, their finish signals are memorised (i.e., latched). The finish signal of the parallel composition is produced only when all the parallel branches have finished (Fig. 3b).

In the case of the selection construct, Fig. 3c, the activation pulse (i.e., start signal) is given to the appropriate sub-circuit only if the selection construct has been started during that clock cycle and if the value of the condition enables its execution. The finish signal of the selection construct is just the logical disjunction of the finish signals of the two sub-circuits. This implementation produces the desired effect provided that the synthesis schema preserves the assumption that a circuit will not generate its finish signal unless it has been previously activated.

The compilation schema for the iteration construct (Fig. 3d) is in principle similar to the one for the selection construct. The most relevant difference is that the finish signal of the while's body is looped back to keep the circuit active after the first iteration over the loop is finished. Also note that if the condition is false, the finish signal is generated combinatorially producing the desired effect of terminating the `while` construct within that clock cycle.

The synthesis of input and output constructs decentralises the control of the communication. Fig. 3e illustrates the hardware generated to perform the actions associated with input/output. The circuit begins the clock cycle by requesting to communicate (i.e., by means of showing its *readiness* to communicate). This signal is processed by a mechanism arbitrating the communication that will respond over the *transfer* line, whether the communication can be performed or not. In the



(a) Sequential composition. (b) Parallel composition. (c) Selection.

(d) While loop. (e) Input/output control. (f) Communication arbitration.

Fig. 3. Circuit templates for the compilation of compound constructs.

former case, the construct behaves like assignment does. In the latter, the circuit memorises its active state and tries again during the next clock cycle.

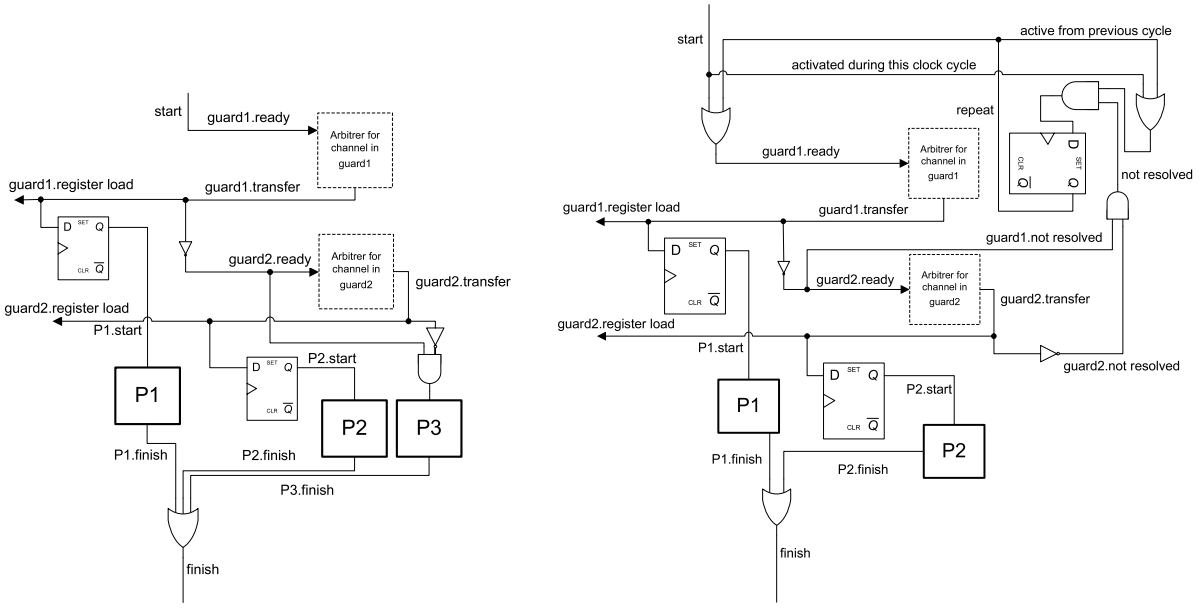
Finally, there is another circuit that is generated by the synthesis process (once per channel) and implements the arbitration mechanism that governs the communications over it (Fig. 3f). The circuit collects all the input and output requests over the channel and it allows the communication when there is at least one input and one output request over the channel during the current clock cycle.

2.2. PriAlt

Up to this point we have presented the compilation schema described in [20] for the constructs in Handel-C ranging from assignment and `delay` to parallel composition and communication. In this subsection we present our extension to this work that allows the synthesis of hardware implementing the `priAlt` construct.

As observed in Section 2, the behaviour (and algebraic properties) of the `priAlt` construct change significantly according to whether it contains a *default* guard or not. We have devised a compilation schema for each of these two possible behaviours. Fig. 4a shows the circuit generated for a `priAlt` construct with default guard of the form **priAlt {case** $guard_1$**:** P_1 ; **break**; **case** $guard_2$**:** P_2 ; **break**; **default:** P_3**}**. The start signal is used as the communication request associated with $guard_1$ sent to the appropriate arbitration circuit. The *transfer* signal returned by the arbiter is used to activate the communication and P_1 after one clock cycle. The negation of the *transfer* signal is used to trigger the communication request for $guard_2$ (in a similar way to how *start* was used for $guard_1$). The same mechanism as described above is used to handle the *transfer* signal for $guard_2$. On the other hand, if all the arbiters return a false *transfer* signal, the circuit associated with P_3 is activated. Finally, the finish signal for the `priAlt` construct is the disjunction of the individual finish signals of P_1, P_2 and P_3. Note that the finish signal is properly generated provided that P_1, P_2 and P_3 produce a *finish* pulse only if they have been started.

The compilation of the `priAlt` construct without a default guard is described in Fig. 4b (here we show the compilation of **priAlt {case** $guard_1$**:** P_1 ; **break**; **case** $guard_2$**:** P_2 ; **break}**). In the case of a `priAlt` with a default guard we activated P_3 when neither $guard_1$ nor $guard_2$ were allowed to communicate. Here we use the same condition to determine whether the `priAlt` has been resolved or not. If it has not been resolved and the circuit is active (i.e., the start signal was given during



(a) Default `priAlt`.

(b) Non-default `priAlt`.

Fig. 4. Circuit templates for the compilation of `priAlt`.

the current clock cycle or the circuit has been started in a previous clock cycle but it has not been able to communicate up to the present clock cycle) this 'active' status is memorised. The *active* state is used to restart the circuit in the next clock cycle.

3. The semantic domain

The existing denotational semantics for the synthesis process [4,21] are based on the notion of "branching sequences" (also referred to as "computation trees" [2] and, in essence, similar to the multi-level lists used by Lengauer [13]). In this setting, non-branching nodes denote the execution of actions in which the control flow of the program does not get modified. Branching nodes, on the other hand, model a decision point (such as the condition in a `while` construct), where the state must be accessed in order to evaluate the condition. The actual *trace* of the program is obtained by keeping track of the updates over the environment and by pruning the branches that do not get executed. Even though this semantic model was successfully implemented and tested against existing semantics for Handel-C [4,5], our goal is to achieve a higher degree of confidence in it by means of proving its correctness.

When trying to prove correctness properties for the compiled hardware, we observed that the semantics fails to capture the hardware parallelism at the branching sequence level (parallel actions performed combinatorially in the hardware are "linearised" in the branching sequences). The main drawback of this feature of the semantics is that most of the properties we are interested in proving hold at the end of clock cycles. In this context, proving relatively simple properties (such as that parallel composition is commutative if only state updates are considered[2]) showed itself to be very complicated, given the need to establish the equivalence at synchronous points (i.e., clock cycle edges) that did not occur at the same depth in the different parallel branches.

To overcome this problem, we observed that Handel-C's synchronous time model allowed us to group the actions performed in each clock cycle into two disjoint sets: *combinatorial actions* (performed before the end of the clock cycle) and *sequential actions* (performed at the end of the clock cycle). The idea of grouping similar actions together is not fully compatible with the tree-like structure used in branching sequences. In particular, branching conditions are combinatorial actions and we want to include them in the corresponding set of actions, rather than having them branching the sequences' structure. We address this problem by formulating branching sequences of the form

$$S \frown cond \rightarrow S_1 \mid S_2$$

(where $\frown$ stands for the concatenation operation over $\mathtt{Seq}$ and $cond \rightarrow S_1 \mid S_2$ represents a branching node that selects S_1 or S_2 depending on whether condition *cond* holds) and expanding them into two sequences:

$$S \frown cond_\perp : S_1 \quad \text{and} \quad S \frown \neg cond_\perp : S_2$$

2 The wiring of $c_1 || c_2$ is different than the one for $c_2 || c_1$, but their effect over the state is the same.

(where $cond_{\perp}$ is an assertion stating that *cond* must hold during the current clock cycle and $a : S$ adds combinatorial action a to the head of sequence S). This observation allows us to turn the semantic domain into a set of finite-length linear sequences (this view of the selection construct is consistent with the one used in other formalisms such as [11]).

A similar problem is encountered when constructs are composed in parallel as a structured node was generated in the sequence, pre-empting the actions in the parallel branches to be collected together. We overcome this problem by introducing a merge operator that joins the pair of sequences being composed in parallel.

Our new semantic domain is the powerset of sequences of the type:

Definition 3.0.1.

$$
\begin{array}{lll}
\texttt{Seq} ::= \texttt{Empty} & & \\
\quad | \ \texttt{Cb}\ \mathscr{P}(\texttt{Action}) \rightarrow \texttt{Seq} & & \text{combinatorial behaviour} \\
\quad | \ \texttt{Ck}\ \mathscr{P}(\texttt{Action}) \rightarrow \texttt{Seq} & & \text{sequential behaviour.}
\end{array}
$$

The `Action` type captures the notion of actions that the hardware can perform together with assertion-like extensions (in the sense of [8]) in order to allow the verifications we need over the hardware. Let w be a wire identifier, w_1 and w_2 expressions involving wires (i.e., elements of type `wExpression` as defined below), var a variable or channel name, val a value in the corresponding type and *cnd* a boolean condition, then we define the `Action` type as:

Definition 3.0.2.

$$
\begin{array}{lll}
\texttt{Action} ::= \texttt{wExp} & & \\
\quad | \ var \leftarrow val & & \text{the store location } var \text{ gets the value } val \\
\quad | \ cnd_{\perp} & & cnd \text{ must be satisfied in the current clock cycle} \\
\texttt{wExpression} ::= w & & w \text{ is set to the } high \text{ value} \\
\quad \neg w & & w \text{ is set to the } low \text{ value} \\
\quad | \ w_1 \wedge w_2 & & w_1 \text{ and } w_2 \text{ are in } high \\
\quad | \ w_1 \vee w_2 & & w_1 \text{ or } w_2 \text{ are in } high \\
\quad | \ w_1 \leftarrowtail w_2 & & \text{the value of } w_2 \text{ is transferred to } w_1.
\end{array}
$$

3.1. Domain operations

Our semantic domain is going to need two operators in order to be able to manipulate sequences within the semantic function: *concatenation* ($\frown$) and *parallel merge* ($\uplus$).

Regarding concatenation, our initial formulation was a non-standard one due to the possibility of having to concatenate a pair of sequences that *overlap*. Two sequences overlap iff the last and first element (in the first and second sequence respectively) are built from the same constructor (for example the application $(\texttt{Cb}\ a_1) \frown (\texttt{Cb}\ a_2\ S_2)$ does overlap).

In the case of overlapping sequences, the lack of synchronous actions in between the last element of the first sequence and the first element of the second one exposes the fact that no clock cycle finishes between them. In the context of our semantics, this indicates the need to join the extremes of the two sequences and generate a single combinatorial node with the union of the comprised sets of actions (in the previous example, the expected result would be $(\texttt{Cb}\ (a_1 \cup a_2)\ S_2)$). In fact, this is a very common case, given the fact that circuits generated for all constructs start and finish their execution performing combinatorial actions, leading to sequences in which the first and last elements are of combinatorial type.

We use a trivial extension of the standard concatenation operator ($\frown$) that is capable of handling our heterogeneous sequences. This definition allows consecutive nodes based on the same constructor to be put in sequence and this kind of behaviour is not suitable in our semantics (we would expect the actions in the two consecutive nodes of the same kind to be collected in a single set of actions). We solve this problem by ensuring that the sequences being concatenated using the $\frown$ operator avoid the problematic cases (see Section 4 for further details).

To address the definition of our merge operator we first need to be able to establish when a pair of sequences is *in-Phase*:

Definition 3.1.1.

$$
\begin{array}{l}
\mathit{in\text{-}Phase}(S_1,\ S_2) : \texttt{Seq} \times \texttt{Seq} \rightarrow \mathit{Bool} \\
\qquad \forall S \in \texttt{Seq} \bullet \mathit{in\text{-}Phase}(S, \langle\rangle) \wedge \mathit{in\text{-}Phase}(\langle\rangle, S) \\
\qquad \forall S_1, S_2 \in \texttt{Seq};\ a_1, a_2 \in \mathscr{P}(\texttt{Action}) \bullet \mathit{in\text{-}Phase}((\texttt{Cb}\ a_1 S_1), (\texttt{Cb}\ a_2 S_2)) \\
\qquad \forall S_1, S_2 \in \texttt{Seq};\ a_1, a_2 \in \mathscr{P}(\texttt{Action}) \bullet \mathit{in\text{-}Phase}((\texttt{Ck}\ a_1 S_1), (\texttt{Ck}\ a_2 S_2)).
\end{array}
$$

In the case of having not *in-Phase* sequences (i.e. a pair of sequences in which the structure is not node-wise equal), the parallel-merge operator should give priority to combinatorial actions over sequential ones (we should allow a combinatorial action to be executed in the current clock cycle if one of the circuits being composed in parallel needs to do so). A typical way of implementing this kind of behaviour is through priorities. Although an efficient solution in terms of implementation, dealing with priorities makes the definition of the function quite complicated and, as in the case of the concatenation function, reduces the set of properties of the function to a minimal one. On the other hand, the definition of the operator becomes completely symmetric (and there is no longer a need for priorities) if we can ensure that the sequences are *in-Phase*:

Definition 3.1.2.

$$
\begin{aligned}
&S_1 \uplus S_2 : \mathtt{Seq} \times \mathtt{Seq} \to \mathtt{Seq}\\
&\textbf{pre } \mathit{in\text{-}Phase}(S_1, S_2)\\
&\quad S_1 \uplus \langle\rangle = S_1\\
&\quad \langle\rangle \uplus S_2 = S_2\\
&\quad (\mathtt{Cb}\ a_1\ S_{1.tail}) \uplus (\mathtt{Cb}\ a_2\ S_{2.tail}) = (\mathtt{Cb}\ a_1 \cup a_2\ (S_{1.tail} \uplus S_{2.tail}))\\
&\quad (\mathtt{Ck}\ a_1\ S_{1.tail}) \uplus (\mathtt{Ck}\ a_2\ S_{2.tail}) = (\mathtt{Ck}\ a_1 \cup a_2\ (S_{1.tail} \uplus S_{2.tail})).
\end{aligned}
$$

3.2. Fix-points

As our semantic function is going to use recursive equations, it is necessary to assure the existence of an appropriate semantic domain with fix-point solutions to them. The general idea is to establish a semantic domain with an associated *complete partial order* (CPO), that is, a set with a partial ordering v, a least element $\bot$, and limits of all non-empty chains [7] and to describe the semantics in terms of continuous functions (i.e., functions between CPOs that preserve the partial order and limit structure).

To achieve this goal we extend Seq with a bottom element $\bot$ and order our lifted semantic domain $\mathtt{Seq}_\bot$ by the relation $\preceq$ below. We prove this ordering to be a partial order and that the constructors are monotonic with respect to it.

Definition 3.2.1.

$$
\begin{aligned}
&\bot \preceq S \wedge \langle\rangle \preceq \langle\rangle \wedge\\
&(\mathtt{Cb}\ a\ S_1) \preceq (\mathtt{Cb}\ a\ S_2) \Leftrightarrow S_1 \preceq S_2 \wedge\\
&(\mathtt{Ck}\ a\ S_1) \preceq (\mathtt{Ck}\ a\ S_2) \Leftrightarrow S_1 \preceq S_2.
\end{aligned}
$$

We have now to verify that the concatenation function is also monotonic with respect to $\preceq$. We first extend $^\frown$ to treat $\bot$ as left and right zero. With this extended definition, we can easily prove $^\frown$ to be monotonic on its first and second arguments (by structural induction on s_1, s_2 and s). We extend the parallel-merge operator to treat $\bot$ as zero when it appears in any of its arguments. We also use $\bot$ as the result for the function when the arguments are not *in-Phase*, as we need to totalise the functions in order to be able to encode them in the HOL theorem prover. The proof of right monotonicity for $\uplus$ is carried out by case analysis on the result of $\uplus$'s application (after using the right kind of induction).

The proof of $\uplus$'s left monotonicity, on the other hand, cannot be performed by structural induction because the sequences cannot be handled as a pair (the induction principle must be applied to individual sequences in a sequential way). In particular, structural induction over the individual sequences provides us with a pair of hypotheses that refer to sequences that are not *in-Phase* with respect to each other (this happens because of the sequential way in which we apply the induction principle). As mentioned earlier in the paper, this is a case in which $\uplus$ is undefined and it only arises because the induction is not set up in the right way.

The solution is to make the proof by complete induction over the sum of the lengths of the sequences being merged. This way of proving the theorem is quite laborious, but allows us to instantiate the inductive hypothesis to the sequences of the right shape when needed. Following this approach combined with the case analysis mentioned above, we prove $\uplus$ left monotonic. We then use this result, together with the fact that $\uplus$ is commutative, to prove that $\uplus$ is also right monotonic.

Having shown a suitable partial order over the semantic domain and proved that all the operators preserve that ordering, we can guarantee that fix-point solutions to the recursive equations introduced in the next section exist in our model.

4. A model for the compilation and its semantics

As we intend to reason about the compilation process from our subset of Handel-C into hardware components we need a way of introducing the language constructs in our model. Moreover, the kind of reasoning we intend to do will imply reasoning over all possible combinations of the constructs in the language. The rest of this section is dedicated to showing the way in which we embedded the input language in HOL and how we used the semantic domain defined in the previous section to give semantics to its compiled forms.

4.1. Embedding Handel-C's syntax in HOL

Boulton et al. [1] provided evidence that a deep embedding of the input language into the model is the best alternative for reasoning about properties regarding programs that can be generated from that input language. Our problem fits this kind of reasoning as we want to analyse the correctness of the hardware generated for all possible (and valid) combinations of our input language.

On the basis of the above observation, we have encoded the input language as recursive datatypes in HOL. In fact, the encoding using recursive datatypes is enough for modelling most of the input language with the exception of the `priAlt` construct. The variability in the length of the guarded commands inside a `priAlt` construct makes it impossible to capture

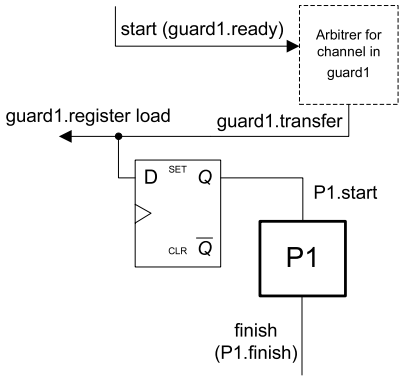


Fig. 5. `priAlt` guard with *start*/*finish* interface.

it as a standard datatype as this would require having a mechanism for defining constructors of variable arity. We solved this problem by defining the recursive type of guarded choices, allowing us to move the variability in length from the `priAlt` construct into its argument. In this way, we can embed the `priAlt` construct as a constructor of a single, but variable in length, argument. From the mechanisation point of view, as the definitions of the datatype describing the constructs in the language and the one describing the list of alternatives inside occurrences of the `priAlt` operator depend on each other, we have implemented them by means of a mutually recursive definition. Our encoding of the input language in HOL is as shown in Definition 4.1.1.

Definition 4.1.1.

$$
\begin{aligned}
\texttt{lang} ::= \;& \text{Delay} : \mathit{wireId} \times \mathit{wireId} \\
| \;& \text{Assign} : \mathit{wireId} \times \mathit{wireId} \times \mathit{varId} \times \mathit{expr} \\
| \;& \text{SeqComp} : \mathit{wireId} \times \mathit{wireId} \times \texttt{lang} \times \texttt{lang} \\
| \;& \text{ParComp} : \mathit{wireId} \times \mathit{wireId} \times \texttt{lang} \times \texttt{lang} \\
| \;& \text{Selection} : \mathit{wireId} \times \mathit{wireId} \times \mathit{cond} \times \texttt{lang} \times \texttt{lang} \\
| \;& \text{While} : \mathit{wireId} \times \mathit{wireId} \times \mathit{cond} \times \texttt{lang} \\
| \;& \text{Input} : \mathit{wireId} \times \mathit{wireId} \times \mathit{chanId} \times \mathit{varId} \\
| \;& \text{Output} : \mathit{wireId} \times \mathit{wireId} \times \mathit{chanId} \times \mathit{expr} \\
| \;& \text{PriAlt} : \mathit{wireId} \times \mathit{wireId} \times \texttt{guardList} \\
\texttt{guardList} ::= \;& \text{List} : \mathit{wireId} \times \mathit{wireId} \times \mathit{guard} \times \texttt{lang} \times \texttt{guardList} \\
| \;& \text{Leaf} : \mathit{wireId} \times \mathit{wireId} \times \mathit{guard} \times \texttt{lang} \\
| \;& \text{Default} : \mathit{wireId} \times \mathit{wireId} \times \texttt{lang}.
\end{aligned}
$$

It is important to notice that we have used the same *start*/*finish* interface from Page's compilation approach [19] to describe the guards inside the `priAlt` construct. We have not described our synthesis schema for `priAlt` in these terms but this problem is easily overcome if we consider all the hardware generated from the *ready* signal up to the *finish* signal of the circuit associated with each guard. Fig. 5 illustrates the idea by showing a projection of the components of Fig. 4a that implement the first guarded alternative and illustrating which wires are used to interface this sub-circuit.

4.2. Denotational semantics for the compilation: the semantic predicate

On the basis of the sequence-based domain defined in the previous section we can give semantics to the translation for all the constructs in our input language. The semantics is going to be described as a set containing all possible execution traces for the program being synthesised. However, due to the presence of circuits with loop-back connections and the fact that our hardware model will capture the semantics of iterating constructs by means of successive syntactic approximations, we cannot apply any form of explicit description of the semantic set. This observation is particularly relevant if we consider that we are also aiming at mechanically verifying our approach. The obvious solution is to find a predicate *smPred* capturing the semantics of the hardware generated from Handel-C program c such that we can define the semantic function $\mathtt{Sm}$ as

$$(\mathtt{Sm}\ c) = \{s : \mathtt{Seq}_{\perp} \mid \mathit{smPred}(c, s)\}.$$

In this context, *inductively defined relations* [18,6] are predicates defined by a set of rules for generating their elements. Inductive definitions can be regarded as a "constructive mechanism" in the sense that they provide information about how to generate some basic elements (base cases) and additional rules that instruct on how to obtain new elements from existing ones.

Following [16,17] we adopt this approach to define our semantic predicate, taking the (informal) rules in the compilation schema as the basis for defining the set of rules for our semantic predicate *smPred*. In particular, `delay`, assignment, `while`

(false case) and successful `input`/`output` are "base case" constructs for the semantics (given the fact that their semantics do not involve the semantics of other syntactic elements). The remaining constructs, on the other hand, involve the semantic expressions for their components and can be regarded as "inductive cases".

The case of the `priAlt` construct introduces an additional level of complexity to the problem as we have to provide semantics to the hardware that is generated for its list of guarded alternatives. Moreover, the mutual dependency between the embedding of the language constructs and the list of alternatives for the `priAlt` operator forced the semantics to also be mutually dependent. Fortunately, it is possible to define the kind of predicates we need by means of mutual induction [25]. With this idea in mind, we included the new relationship *smGuard* that generates the corresponding semantic expressions for the list of guarded alternatives associated with each occurrence of `priAlt` in a given program. Note that as the list of alternatives inside `priAlt` constructs are not valid program constructs on their own (i.e., they can only occur inside priAlts) this extension of the semantic predicate does not affect our previous definition of the semantic function `Sm`.

We also need a way to incorporate the unique pair of wires (*start* and *finish*) generated for each construct in the program by the synthesis process. We do so by means of two functions ($\pi_s(c)$ and $\pi_f(c)$) returning, respectively, the *start* and *finish* identifiers for a given circuit c.

In our semantics, the hardware components generated by the synthesis process start their execution by performing a set of combinatorial actions (essentially to propagate their start signal to their constituent constructs) and also finish by carrying out a set of combinatorial actions (to propagate the finish signal appropriately). Furthermore, these "before" and "after" combinatorial actions performed by all the circuits are likely to be executed during the same clock cycle in which the previous circuit was terminating (or the next one is starting). This suggests that these are points in which the merging of action sets should take place to condense actions of the same type that happen in the same clock cycle into a single node.

We capture this notion by isolating these special points in the semantics, allowing us to have greater control over the structure of the sequences and the way in which they get concatenated/merged. We redefine our semantic predicate to relate a circuit c with two sets of combinatorial actions: *prologue* and *epilogue* (accounting respectively for actions at the beginning and end of the execution of the circuit) and a *behavioural sequence* (capturing the actions being executed in between these two combinatorial fragments).

On the other hand, the fact that the actions when the `while` construct terminates and default actions in a `priAlt` reduce to pure combinatorial actions makes their behaviour different from the rest of the constructs (that take at least one clock cycle to finish their execution). In this sense, we allow *smPred* to produce two kinds of results: *mixed*, to capture the semantics of constructs involving both combinatorial and sequential actions (this is our formulation based on the prologue, behavioural sequences and epilogue mentioned above); and *combinatorial*, to encode the case in which the semantics involve only actions that are performed within the current clock cycle. We split the actions in this last type of semantic expression into two, allowing us to have a prologue and epilogue when we produce the combinatorial type of result. We formalise this notion by defining the following result type:

Definition 4.2.1.

$$\begin{aligned}\texttt{smResult} ::= \;& \texttt{Mixed}\ \mathscr{P}(\mathit{Action}) \to \texttt{Seq} \to \mathscr{P}(\mathit{Action}) \\ \mid\; & \texttt{Combin}\ \mathscr{P}(\mathit{Action}) \to \mathscr{P}(\mathit{Action}).\end{aligned}$$

For the remainder of the paper, we will use $\lceil\ \rceil$ and $[\![\ ]\!]$ as syntactic shorthand for the `Mixed` and `Combin` constructors respectively. In these terms, the semantics of the `delay` construct state that its combinatorial prelude only includes a verification for the *start* wire, while its combinatorial epilogue just sets its finish wire to the high value. The behavioural part of the circuit just states that it delays its execution for a single clock cycle (Definition 4.2.2).

Definition 4.2.2.

$$\forall s, f \in \mathit{wireId} \bullet \mathit{smPred}\big((\texttt{delay}\ s\ f), \lceil\ \{s_\perp\}\ (\texttt{Ck}\ \{\mathit{skip}\})\ \{f\}\ \rceil\big).$$

The semantics for the assignment construct is very similar but the behavioural component is modified to capture the update to the store:

Definition 4.2.3.

$$\forall s, f \in \mathit{wireId}; x \in \mathit{varId}; e \in \mathit{expr} \bullet \mathit{smPred}\big((\texttt{Assign}\ s\ f\ x\ e), \lceil\{s_\perp\}\ (\texttt{Ck}\ \{x \leftarrow e\})\ \{f\}\ \rceil\big).$$

In the case of constructs c_1 and c_2 being sequentially composed, the prelude transfers the *start* signal from the sequential composition circuit to c_1's *start* and also includes c_1's combinatorial prelude (notice here that we are joining at this point the sequential composition's and c_1's preludes). The behavioural part comprises c_1's behaviour followed by a combinatorial set of actions turning the finish signal of c_1 into c_2's start and performing c_2's prelude, to conclude with c_2's behaviour. Finally, the sequential composition's epilogue is composed of c_2's epilogue and combinatorial hardware to propagate c_2's finish signal as the sequential composition's one. Definition 4.2.4 presents a more formal description of these actions.

Definition 4.2.4.

$$\begin{array}{l}\forall\, s, f \in wireId;\ c_1, c_2 \in \texttt{lang};\ init_{c_1}, link_{c_1}, init_{c_2}, link_{c_2} \in \mathscr{P}(\texttt{Action});\ seq_{c_1}, seq_{c_2} \in \texttt{Seq} \bullet \\ \quad smPred\big(c_1, \big[init_{c_1}\ seq_{c_1}\ link_{c_1}\big]\big) \land smPred\big(c_2, \big[init_{c_2}\ seq_{c_2}\ link_{c_2}\big]\big) \Rightarrow \\ \qquad smPred\big((\texttt{Scomp}\ s\ f\ c_1\ c_2), \\ \qquad\qquad \big[init_{c_1} \cup \{s_\perp ; \pi_s(c_1) \leftarrowtail s\} \\ \qquad\qquad\ seq_{c_1} \frown (\texttt{Cb}\ (link_{c_1} \cup \{\pi_s(c_2) \leftarrowtail \pi_f(c_1)\} \cup init_{c_2})\ seq_{c_2}) \\ \qquad\qquad\ (link_{c_2} \cup \{f \leftarrowtail \pi_f(c_2)\})\big]\big).\end{array}$$

It is easy to observe from the definition above that there is no assurance (at least, not at the syntactic level) that the precondition we established for the concatenation function is going to be satisfied. In particular, we need to guarantee that all possible behavioural sequences (as we could essentially compose any construct in sequence with any other) begin and end with a `Ck` node. As we explained before, this is why it is not possible to allow empty behavioural sequences (as needed to describe the semantics for the case of a terminating `while` construct). In Section 5 we define a subclass of our sequences in which the required property holds and prove that all possible behavioural sequences produced by our semantic predicate (i.e., when the `Mixed` type of result is produced) belong to that subclass.

We also need to add rules to handle the cases in which one of the constructs (or both) being composed sequentially only performs combinatorial actions. As an example, the case in which the first construct terminates "immediately" is described by Definition 4.2.5. For conciseness, for the rest of the paper we omit the domain of quantified variables when this information is clear from the context.

Definition 4.2.5.

$$\begin{array}{l}\forall\, s, f, c_1, init_{c_1}, link_{c_1}, c_2, init_{c_2}, seq_{c_2}, link_{c_2} \bullet \\ \quad smPred(c_1, \llbracket init_{c_1}\ link_{c_1} \rrbracket) \land smPred\big(c_2, \big[init_{c_2}\ seq_{c_2}\ link_{c_2}\big]\big) \Rightarrow \\ \qquad smPred\big((\texttt{Scomp}\ s\ f\ c_1\ c_2), \\ \qquad\qquad \big[init_{c_1} \cup \{s_\perp ; \pi_s(c_1) \leftarrowtail s\} \cup link_{c_1} \cup \{\pi_s(c_2) \leftarrowtail \pi_f(c_1)\} \cup init_{c_2} \\ \qquad\qquad seq_{c_2} \\ \qquad\qquad (link_{c_2} \cup \{f \leftarrowtail \pi_f(c_2)\})\big]\big).\end{array}$$

The symmetric case (the second construct terminates within the same clock cycle in which it was started) is described in a similar way and we do not show it here. The case in which the two constructs being composed in sequence terminate in the same clock cycle also terminates in a single clock cycle (Definition 4.2.6).

Definition 4.2.6.

$$\begin{array}{l}\forall\, s, f, c_1, init_{c_1}, link_{c_1}, c_2, init_{c_2}, link_{c_2} \bullet \\ \quad smPred(c_1, \llbracket init_{c_1}\ link_{c_1} \rrbracket) \land smPred(c_2, \llbracket init_{c_2}\ link_{c_2} \rrbracket) \Rightarrow \\ \qquad smPred((\texttt{Scomp}\ s\ f\ c_1\ c_2), \\ \qquad\qquad \llbracket init_{c_1} \cup \{s_\perp ; \pi_s(c_1) \leftarrowtail \pi_s(s)\} \cup link_{c_1} \cup \{\pi_s(c_2) \leftarrowtail \pi_f(c_1)\} \cup init_{c_2} \\ \qquad\qquad link_{c_2} \cup \{f \leftarrowtail \pi_f(c_2)\} \rrbracket).\end{array}$$

It is important to notice that none of the three additional rules generated to handle the case of a *combinatorial* result needs to apply the concatenation operation to produce its results. We will take advantage of this observation when proving the correctness in the application of the domain operators within the semantic predicate.

In the case of the parallel composition of c_1 and c_2, the combinatorial prelude propagates the parallel composition's *start* signal to c_1 and c_2 *start* wires and brings together their combinatorial preludes. The behavioural component of the semantics is just constructed by parallel merging c_1 and c_2's behavioural sequences. Finally, the actions in the epilogue include the epilogues of both c_1 and c_2, together with combinatorial logic to generate the finish signal for the parallel composition when $\pi_f(c_1)$ and $\pi_f(c_2)$ are in *high* (Definition 4.2.7).

Definition 4.2.7.

$$\begin{array}{l}\forall\, s, f, c_1, init_{c_1}, seq_{c_1}, link_{c_1}, c_2, init_{c_2}, seq_{c_2}, link_{c_2} \bullet \\ \quad smPred\big(c_1, \big[init_{c_1}\ seq_{c_1}\ link_{c_1}\big]\big) \land smPred\big(c_2, \big[init_{c_2}\ seq_{c_2}\ link_{c_2}\big]\big) \Rightarrow \\ \qquad smPred\big((\texttt{Pcomp}\ s\ f\ c_1\ c_2), \big[init_{c_1} \cup init_{c_2} \cup \{s_\perp ; \pi_s(c_1) \leftarrowtail s ; \pi_s(c_2) \leftarrowtail s\} \\ \qquad\qquad\qquad\qquad seq_{c_1} \uplus seq_{c_2} \\ \qquad\qquad\qquad\qquad (link_{c_1} \cup link_{c_2} \cup \{f \leftarrowtail \pi_f(c_1) \land \pi_f(c_2)\})\big]\big).\end{array}$$

When defining the semantics of parallel composition, the second simplification in our hardware model makes itself evident: we are not using flip-flops to memorise the finish token of the circuit with the shortest execution in order to preserve its value until the other parallel circuit has also finished. Even though it would be possible to modify the definition of $\uplus$ to account for this, we opted to not follow this idea for the following reasons:

- The flip-flop implementation in the actual hardware is a good solution at the circuit level for ensuring that no finish signal will be lost while composing circuits of different length in parallel. We achieve the same effect at the semantic level by making the resulting composed sequence finish only when both sequences have finished and by making the information that both c_1 and c_2 have finished available for the combinatorial actions following that behavioural sequence.
- The compilation definition does not allow any wire to connect directly to the individual finish wires of either c_1 or c_2 (these wires are, in fact, "hidden" within the wrapper constructed by the parallel composition structure). This fact implies that there are no correctness concerns that we need to address from the possible connections from these wires (we can be certain, for example, that no circuit will use the finish pulse of c_1 to activate itself, as it cannot access it), allowing us to safely abstract out this feature in our semantic model.

As with the sequential composition construct, it is also necessary to address the cases involving the instantaneous termination of the constructs being composed in parallel. We omit these rules (together with the ones handling the selection construct) as they are similar to the rules we have already presented. The complete set of rules can be found in [22].

The semantics for the `while` construct needs to provide rules for handling the two possible outcomes of the evaluation of the condition. The first rule accounts for the case when the condition is false and the `while` terminates immediately (Definition 4.2.8).

Definition 4.2.8.

$$\forall s, f, c, body \bullet smPred((\texttt{While}\, s\, f\, c\, body), [\![\{s_\perp ; (\neg c)_\perp\}\{f\}]\!]).$$

When the condition holds, the traditional notion of syntactic approximations [26] is applied. We provide two rules for the approximation: a base case capturing the first approximation to the solution (by means of a single execution of the `while`'s body) and another one to capture the way in which we can construct a longer approximation from an existing one.

The first approximation calculates the semantics of the while's body, assumes that the looping condition does not hold and uses its epilogue to signal the `while`'s termination (Definition 4.2.9).

Definition 4.2.9.

$$\begin{array}{l}\forall s, f, b, init_b, seq_b, link_b, c \bullet \\ \quad smPred(b, [init_b\ seq_b\ link_b]) \Rightarrow \\ \quad\quad smPred((\texttt{While}\, s\, f\, c\, b), [\{s_\perp ; c_\perp ; \pi_s(b) \hookleftarrow s\} \cup init_b \\ \quad\quad\quad\quad\quad\quad seq_b \\ \quad\quad\quad\quad\quad\quad \{s \hookleftarrow \pi_f(b)\} \cup link_b \cup \{\neg c_\perp ; s_\perp ; f \hookleftarrow (s \wedge \neg c)\}]).\end{array}$$

Following the approach we have been using so far, it would be natural to add a rule to handle the case in which the `while`'s body finishes immediately. This situation corresponds, for example, to the C program: `while (1) {while (0);}`. This case is regarded as "pathological" [4] as the construct *stutters* without making any progress or advancing the clock. In fact, all implementations of the Handel-C compiler spot this case and flag the program as incorrect. As this problem has been already detected (and corrected from the implementation's perspective), we decided to exclude it from our analysis. In this sense, we do not contemplate the possibility of a `while`'s body terminating immediately and, hence, we do not add rules for handling it.

The final rule for the `while` construct is meant to extend an existing approximation (generated either by the basic rule above or by previous applications of itself). The approximation is constructed by appending one expansion of the body and the proper linking combinatorial action in front of the approximation's behavioural sequence (Definition 4.2.10).

Definition 4.2.10.

$$\begin{array}{l}\forall s, f, b, init_b, seq_b, link_b, init_{approx}, seq_{approx}, link_{approx}, c \bullet \\ \quad smPred(b, [init_b\ seq_b\ link_b]) \wedge \\ \quad smPred((\texttt{While}\, s\, f\, c\, body), [init_{approx}\ seq_{approx}\ link_{approx}]) \Rightarrow \\ \quad\quad smPred((\texttt{While}\, s\, f\, c\, b), \\ \quad\quad\quad\quad [init_{approx} \\ \quad\quad\quad\quad seq_{approx} \frown (\texttt{Cb}\ \{c_\perp ; s_\perp ; \pi_s(b) \hookleftarrow s ; s \hookleftarrow \pi_f(b)\} \cup link_{approx} \cup init_b\ seq_b) \\ \quad\quad\quad\quad link_{approx}]).\end{array}$$

We use the approximation-based principle defined above to give semantics to the `input` and `output` primitives. In this particular case, we need two rules per constructor because the approximation part of `inputs` and `outputs` only requires waiting for a clock cycle and retrying the communication (the equations for while are more complicated given the fact that we require including the body's semantics on each approximative step). In providing the semantics for `input` and `output`, we need to address the different parts involved in the communication. We use *ch*?, *ch*! and *ch* to denote the reading, writing and bus (the physical means by which the value being communicated is transmitted) components in the communication over channel *ch*. Note that *ch*?, *ch*! represent the corresponding *ready* wires used in the compilation approach described in Fig. 3e.

The base case for the inputting construct uses its prelude to state the presence of the reading component and requests the presence of the writing part (by means of an assertion). If this is the case, the behavioural part of the semantics updates the store according to the value being transmitted over the channel. The actions in the epilogue are used to establish the generation of the finish signal (Definition 4.2.11).

Definition 4.2.11.

$$\forall s, f, ch, x \bullet smPred\big((\texttt{Input}\ s\ f\ ch\ x), \big[\{s_\perp; ch?\,; ch!_\perp\}\ (\texttt{Ck}\ \{x \leftarrow ch\})\ \{f\}\big]\big).$$

The rule defining the approximations when the writer is not ready to communicate just asserts the proper condition in the combinatorial prelude ($\neg ch!_\perp$) and adds a delay of one clock cycle and appropriate combinatorial actions in front of the existing approximation (Definition 4.2.12).

Definition 4.2.12.

$$\begin{array}{l}\forall s, f, ch, x, init_{approx}, seq_{approx}, link_{approx}\bullet\\ \quad smPred\big((\texttt{Input}\ s\ f\ ch\ x), \big[init_{approx}, seq_{approx}, link_{approx}\big]\big) \Rightarrow\\ \quad\quad smPred\big((\texttt{Input}\ s\ f\ ch\ x),\\ \quad\quad\quad \big[init_{approx}\ (\texttt{Ck}\ \{skip\})\ (\texttt{Cb}\ (init_{approx} \cup \{s\})\ seq_{approx})),\ link_{approx}\big]\big).\end{array}$$

Note that all the approximations constructed by this rule conclude with a successful communication (to see why, consider that the base case for the communication primitives is a successful one, and the fact that all approximations will just add "failing" communication behaviour in front of this successfully terminating trace). This fact is the key factor we use to order approximations (see Section 3.2) as the "successful termination" part of the sequence is turned into $\perp$ by the prune function should the construct not terminate its execution at this moment of execution (i.e., the approximation has not yet reached the fix-point solution and, hence, the assertion stating the terminating condition to hold cannot be satisfied).

Finally, the base case for the `output` construct is similar to the one for `input`, but it inverts the roles in the combinatorial prelude (it establishes the presence of the writer and asserts the readiness of the reader). Its behavioural part is also different as it has to assign the value being transmitted to the appropriate channel (Definition 4.2.13).

Definition 4.2.13.

$$\forall s, f, ch, e \bullet smPred\big((\texttt{Output}\ s\ f\ ch\ e), \big[\{s_\perp; ch!\,; ch?_\perp\,; ch \leftarrow e\}\ (\texttt{Ck}\ \{skip\})\ \{f\}\big]\big).$$

The rule capturing the inductive approximations to the `output` construct using the same approach as was introduced for the `input` approximative solution is as described by Definition 4.2.14.

Definition 4.2.14.

$$\begin{array}{l}\forall s, f, ch, e, init_{approx}, seq_{approx}, link_{approx}\bullet\\ \quad smPred\big((\texttt{Output}\ s\ f\ ch\ e), \big[init_{approx}\ seq_{approx}\ link_{approx}\big]\big) \Rightarrow\\ \quad\quad smPred\big((\texttt{Output}\ s\ f\ ch\ e),\\ \quad\quad\quad \big[init_{approx}\ (\texttt{Ck}\ \{skip\})\ (\texttt{Cb}\ (init_{approx} \cup \{s\})\ seq_{approx}))\ link_{approx}\big]\big).\end{array}$$

Having formulated the semantic predicate *smPred*, we still need to provide a way to use it in the definition of our semantic function `Sm`. Considering the fact that the prelude and epilogue components of the semantic predicate are just sets of combinatorial actions, we define the semantic function for a given syntactic construct c as follows:

Definition 4.2.15.

$$\begin{array}{l}\texttt{Sm}\ c = \{s \mid \exists\ init_c, seq_c, link_c \bullet\\ \quad \big(smPred\big(c, \big[init_c\ seq_c\ link_c\big]\big) \Rightarrow (s = (\texttt{Cb}\ init_c\ seq_c)\ ^\frown\ (\texttt{Cb}\ link_c))\big) \wedge\\ \quad (smPred(c, [\![init_c\ link_c]\!]) \Rightarrow (s = (\texttt{Cb}\ init_c \cup link_c)))\}.\end{array}$$

4.2.1. Semantics for the priAlt construct

In order to give semantics to the `priAlt` construct we first need to describe how the semantic predicate for the list of guarded alternatives works. As mentioned in previous sections, we have included the predicate *smGuard* to account for the semantic effects of the different guards associated with a `priAlt` construct. Moreover, the semantics of the `priAlt` as a whole is dependent on whether any of the guarded alternatives has succeeded in communicating (i.e., the `priAlt` has

been *resolved*) or not. With this idea in mind, the *smGuard* relates a list of guarded alternatives not only with a semantic expression but also with a boolean value reflecting whether the `priAlt` was resolved or not. The signature of the *smGuard* predicate is

$$\forall G \in \texttt{guardList}; S \in \texttt{smResult}; res \in bool \bullet smGuard(G, S, res)$$

where G corresponds to the list of guards which we are calculating the semantics for, S are the semantics associated with the possible evaluations of G, and *res* indicates whether the `priAlt` has been resolved (i.e., any of the communications succeeded or a default guard was reached) or not. *smGuard* is also an inductively defined predicate. Its definition is given by means of the different induction cases presented below.

Definition 4.2.16 shows the semantic expression associated with the last guarded command when the communication has not been possible (i.e., the `priAlt` has not been resolved).

Definition 4.2.16.

$$\big(\forall s, f, ch, e, c \bullet smGuard\big((\texttt{Leaf}\ \ s\,f\ \textbf{case}\ ch\textbf{!}e\textbf{:}\ c\ \fatsemi\ \textbf{break}), \langle\{s_\perp\,;\, ch!\,;\, (\neg ch?)_\perp\}\ \{\neg f\}\rangle, F\big)\big).$$

The case of the last guard becoming active when the associated guarded alternative produces mixed results is captured in Definition 4.2.17. The definition that captures the same situation but with the associated guarded fragment produces combinatorial results similar to the one already presented (the sequence part of the result reduces to just $(\texttt{Ck}\ \{skip\})$) and we do not show it here. In the rest of this section we will only show definitions associated with the mixed results given the similarity with their combinatorial counterparts.

Definition 4.2.17.

$$\begin{array}{l}\big(\forall s, f, ch, e, c, init_c, seq_c, link_c \bullet smPred\big(c, \big[init_c\ seq_c\ link_c\big]\big) \Rightarrow \\ \quad smGuard\big((\texttt{Leaf}\ \ s\,f\ \textbf{case}\ ch\textbf{!}e\textbf{:}\ c\ \fatsemi\ \textbf{break}), \\ \qquad \big[\{s_\perp\,;\ ch!\,;\ ch?_\perp\,;\ ch \leftarrow e\}\ (\texttt{Ck}\ \{skip\})\ (\texttt{Cb}\ \{init_c\})\ seq_c\ ))\ link_c \cup \{f\ \leftarrowtail \pi_f(c)\}\big], T\big)\big).\end{array}$$

The last base case for our inductive definition over guarded alternatives for the `priAlt` construct describes the semantics for the hardware associated with the default clause (Definition 4.2.18).

Definition 4.2.18.

$$\begin{array}{l}\big(\forall s, f, c, init_c, seq_c, link_c \bullet smPred\big(c, \big[init_c\ seq_c\ link_c\big]\big) \Rightarrow \\ \quad smGuard\big((\texttt{Default}\ \ s\,f\ c), \big[init_c\ seq_c\ link_c \cup \{f\ \leftarrowtail \pi_f(c)\}\big], T\big)\big).\end{array}$$

So far we have described the base cases the list of alternatives can have (a singleton list or a default action). The case where we have a list longer than a single element as argument for the `priAlt` uses the previous rules to treat the first element in the list. Definition 4.2.19 illustrates the case where the first guard in the list is allowed to communicate. Note how the epilogue produces the disjunction of the finish signal of the current guarded command with the finish signal of the remaining guarded commands in the list. In this way, we produce the n-way *or* gate in Fig. 4a and b by means of chaining binary *or* gates.

Definition 4.2.19.

$$\begin{array}{l}\big(\forall s, f, c, guard, rest, init_g, seq_g, link_g \bullet \\ \quad smGuard\big((\texttt{Leaf}\ \ s\,f\ guard\ c), \big[init_g\ seq_g\ link_g\big], T\big) \Rightarrow \\ \quad\ smGuard\big((\texttt{List}\ \ s\,f\ guard\ c\ rest), \big[init_g\ seq_g\ link_g \cup \{f\ \leftarrowtail \pi_f(c) \vee \pi_f(rest)\}\big], T\big)\big).\end{array}$$

The next definition covers the case where the first guard does not succeed in communicating but one of the lower priority guards in the list does (Definition 4.2.20). There is, of course, another definition describing the case where neither the first guard nor any of the remaining guards succeed in communicating. The definition of that rule is similar to 4.2.20 (with the expected difference in the boolean value indicating the state of the communication) and we do not show it here.

Definition 4.2.20.

$$\begin{array}{l}\big(\forall s, f, s_1, f_1, c, guard, rest, init_g, link_g, init_{rest}, seq_{rest}, link_{rest} \bullet \\ \quad smGuard((\texttt{Leaf}\ \ s_1\,f_1\ guard\ c), \llbracket init_g\ link_g \rrbracket, F) \wedge \\ \quad smGuard\big(rest, \big[init_{rest}\ seq_{rest}\ link_{rest}\big], T\big) \Rightarrow \\ \qquad smGuard\big((\texttt{List}\ \ s\,f\ guard\ c\ rest), \\ \qquad\qquad \big[\{s_1 \leftarrowtail s\,;\ \pi_f(rest) \leftarrowtail \neg f_1\} \cup init_g \cup link_g \cup init_{rest} \\ \qquad\qquad\ seq_g\ link_{rest} \cup \{f\ \leftarrowtail \pi_f(c) \vee \pi_f(rest)\}\big], T\big)\big).\end{array}$$

From the above definitions it is possible to capture the semantics of the `priAlt` operator. Definition 4.2.21 shows the case of a `priAlt` without a default guard where one of the guards was able to communicate. This definition also describes the case when there is a default clause and it has been activated.

Definition 4.2.21.

$$\begin{array}{l}(\forall s, f, g, init_g, seq_g, link_g \bullet smGuard(g, [init_g\ seq_g\ link_g], T) \Rightarrow \\ \quad smPred((\texttt{priAlt}\ \ s\ f\ g), [\{\pi_s(g) \leftarrowtail s\} \cup init_g\ seq_g\ link_g \cup \{f \leftarrowtail \pi_f(g)\}]))\end{array}$$

On the other hand, the case where none of the guards is able to communicate makes the `priAlt` construct wait for a clock cycle and start again. As with other iterating constructs, we use successive approximations to model the behaviour of the construct in this context. Definition 4.2.22 shows the details of the semantics of this case. Note that the semantics associated with the attempt of communicating during the first clock cycle terminate combinatorially.

Definition 4.2.22.

$$\begin{array}{l}(\forall s, f, g, init_g, link_g, init_{appx}, seq_{appx}, link_{appx} \bullet \\ \quad smGuard(g, [\![init_g\ link_g]\!], F) \wedge smPred((\texttt{priAlt}\ \ s\ f\ g), [init_{appx}\ seq_{appx}\ link_{appx}]) \Rightarrow \\ \quad\quad smPred((\texttt{priAlt}\ \ s\ f\ g), [\{\pi_s(g) \leftarrowtail s\} \cup init_g\ \cup link_g \\ \qquad\qquad\qquad\qquad (\texttt{Ck}\ \{Skip\}\ (\texttt{Cb}\ \{s\} \cup init_{appx}\ seq_{appx})) \\ \qquad\qquad\qquad\qquad link_{appx} \cup \{f \leftarrowtail \pi_f(g)\}]))\end{array}$$

4.3. Pruning

So far we have described the semantics of the translation from Handel-C into net-lists as a (possibly infinite) set of finite-length sequences. In order to complete the semantic description of the generated circuits, we need to find (if it exists) a single sequence that specifies the actual execution path and outcome of the program being synthesised.

As in [21], we define two auxiliary functions:

$$\Delta Env : \texttt{env} \to \texttt{Action} \to \texttt{env} \quad \text{and} \quad flattenEnv : \texttt{env} \to \texttt{env}.$$

The former updates the environment according to the action passed as argument by means of rewriting the appropriate function using λ-abstractions. In the particular case of *skip*, ΔEnv treats it as its unit value and returns the same environment. On the other hand, *flattenEnv* is meant to be used to generate a new environment after a clock cycle edge. It flattens all wire values (to the logical value false), resets the channel values to the undefined value and advances the time-stamp by one unit.

We define the execution of sets of hardware actions by means of the predicate

$$exec : \texttt{env} \times \mathscr{P}(\texttt{Action}) \to (\texttt{env}, \mathscr{P}(\texttt{Action}))$$

by means of the following rules:

$$\frac{s = \{\}}{exec(e, s) = (e, \{\})} \qquad \frac{s \neq \{\} \wedge a \in s}{exec(e, s) = exec(\Delta Env(e, a), s - \{a\})}.$$

We also need to be able to handle assertions, so we introduce the function

$$sat^{\perp} : \texttt{env} \times \mathscr{P}(\texttt{Action}) \to bool$$

defined as

$$sat^{\perp}(e, set) = \forall a \in set \bullet holds(a, e)$$

where $holds(a, e)$ is true iff the assertion a is true in the environment e.

As we are dealing with sets of actions and assertions on each node of our sequences, we need to define the collective effect of this heterogeneous set of actions over the environment. The first difficulty we face when defining how a set of actions is going to be executed is that the initial order between actions and conditions has been lost. This is, however, not a problem if we consider that assertions and control flow conditions refer only to the present value of the memory and all variables preserve their values during the whole clock cycle. This fact makes the evaluation of assertions and control flow decisions independent of the combinatorial actions performed in parallel with them and they can be evaluated at any time.

From the observation above, we split the set of actions into the disjoint sets of assertions (As) and the remaining "unconditional actions" (HAs). The partition into As/HAs is induced over the set of actions by means of the functions ∇_{A} and ∇_{HA}. Also from the observations above, we know that control flow assertions can be evaluated at any time, and that wire-correctness assertions must be evaluated after HAs, allowing us to establish the following order of evaluation: HAs $\prec$ As. Taking advantage of this execution order, we can introduce the function

$$setExec : \texttt{env} \times \mathscr{P}(\texttt{Action}) \to (\texttt{env} \cup \perp)$$

defined by the rule for the successful case:

$$\frac{\forall e, e' \in \mathtt{env}; s \in \mathscr{P}(\mathtt{Action}) \bullet (e', \{\}) = exec(e, \nabla_{\mathrm{HA}}(s)) \land sat^{\perp}(e', \nabla_{\mathrm{A}}(s))}{setExec(e, s) = e'}$$

or otherwise:

$$\frac{\forall e, e' \in \mathtt{env}; s \in \mathscr{P}(\mathtt{Action}) \bullet (e', \{\}) = exec(e, \nabla_{\mathrm{HA}}(s)) \land \neg sat^{\perp}(e', \nabla_{\mathrm{A}}(s))}{setExec(e, s) = \perp}.$$

In turn, the above functions can be used to define a single-node execution function for sequences

$$nodeExec : \mathtt{env} \times \mathtt{Seq} \to \{(\mathtt{env}, \mathtt{Seq}_{\perp})\} \cup \{(\mathtt{env}, \checkmark)\}.$$

The simplest case is successful termination (i.e., when the sequence we are trying to execute is empty), captured by the following rule:

$$\frac{seq = \langle\rangle}{nodeExec(e, seq) = (e, \checkmark)}.$$

The next case captures the execution of a sequence that begins with a set of actions containing unsatisfiable conditions (the symmetric case is similar and we omit it here):

$$\frac{\exists ca \in \mathscr{P}(\mathtt{Action}), s_1 \in \mathtt{Seq} \bullet seq = (\mathtt{Cb}\ ca\ s_1) \land setExec(e, ca) = \perp}{nodeExec(e, seq) = (e, \perp)}.$$

The case in which it is possible to perform all actions and satisfy all tests within a combinatorial node is described by the following rule:

$$\frac{\exists ca \in \mathscr{P}(\mathtt{Action}), s_1 \in \mathtt{Seq} \bullet seq = (\mathtt{Cb}\ ca\ s_1) \land setExec(e, ca) = e_{new}}{nodeExec(e, seq) = (e_{new}, s_1)}.$$

The counterpart of the above rule (dealing with sequences starting with a clock-edged block) is as follows:

$$\frac{\exists ca \in \mathscr{P}(\mathtt{Action}), s_1 \in \mathtt{Seq} \bullet seq = (\mathtt{Ck}\ ca\ s_1) \land setExec(e, ca) = e_{new}}{nodeExec(e, seq) = (flattenEnv(e_{new}), s_1)}.$$

Note that the environment needs to be *flattened* after all actions at the clock edge have taken place. The flattening can take place only at this point because of the possibility of having a value being transmitted over a bus (we will lose the value being transferred if we flatten the environment before updating the store with it).

Having described the mechanism to execute a single node, the extension to executing a complete sequence is captured by the predicate

$$seqExec : \mathtt{env} \times \mathtt{Seq}_{\perp} \to \{\perp, \checkmark\}$$

defined as

$$\begin{array}{l}
\forall e \in \mathtt{env} \bullet seqExec(e, \langle\rangle) = \checkmark \\
\forall e \in \mathtt{env} \bullet seqExec(e, \checkmark) = \checkmark \\
\forall e \in \mathtt{env} \bullet seqExec(e, \perp) = \perp \\
\forall e, e' \in \mathtt{env}; s, s' \in \mathtt{Seq} \bullet (nodeExec(e, s) = (e', s')) \Rightarrow seqExec(e', s').
\end{array}$$

Finally, the actual execution path for the synthesised program (for the case in which the program terminates) is given by the operator

$$prune : \mathtt{env} \to \mathscr{P}(\mathtt{Seq}) \to \mathscr{P}(\mathtt{Seq})$$

defined as

Definition 4.3.1.

$$prune(e, sSet) = \{S \in sSet \mid seqExec(e, S) = \checkmark\}.$$

Note that, as Handel-C's control flow is governed by boolean conditions, only one of the possible branches is executable at any given time, making our semantic traces *mutually exclusive*. From this observation, it follows that only one sequence (if any) exists that will lead to successful termination, leading to the ✓ result in the *seqExec* predicate above. In consequence, the set described in Definition 4.3.1 is either a singleton set, modelling termination, or the empty set (the original program diverges).

5. Consistency considerations

Our definition of the semantic function relies on the sequences produced to have certain properties in order to satisfy the preconditions imposed by the operations in our semantic domain. The concatenation function requires the behavioural sequences produced by *smPred* to have the pattern $(\mathtt{Cb}\ a_1\ s)^\frown(\mathtt{Cb}\ a_n)$ for sets of combinatorial actions a_1 and a_n and a sequence s. Furthermore, we need to ensure that the sequences produced by the semantics are *in-Phase* to ensure a safe application of the merge operator.

The methodology we use to ensure the satisfaction of these requirements is as follows. We first define a suitable subset of the sequences in our semantic domain and prove that the properties mentioned above hold for any element in this subset. Then we prove that all the sequences produced by the semantic predicate belong to this subset, ensuring that the semantic function also satisfies the required conditions.

5.1. Clock-bounded sequences

Given the properties we need to satisfy, we observed that all instances of s above should have their *boundaries* (i.e., first and last elements) based on the Ck constructor. Moreover, s should *alternate* (there shouldn't be two consecutive applications of the same constructor). The predicate *ckBnd* captures this notion by means of an inductive definition:

Definition 5.1.1.

$$(\forall\, cka \in \mathscr{P}(Action) \bullet ckBnd(\mathtt{Ck}\ cka)) \land$$
$$(\forall\, cka, ca \in \mathscr{P}(Action), S \in \mathtt{Seq} \bullet ckBnd(S) \Rightarrow ckBnd(\mathtt{Ck}\ cka\ (\mathtt{Cb}\ ca\ S))).$$

We also need to show that any pair of sequences satisfying the *ckBnd* property is also *in-Phase*. The *alternating* nature of *ckBnd* sequences also guarantees the satisfaction of this requirement, allowing us to prove:

Lemma 5.1.1.

$$\forall S_1, S_2 \in \mathcal{S}eq \bullet ckBnd(S_1) \land ckBnd(S_2) \Rightarrow in\text{-}Phase(S_1, S_2).$$

5.2. The semantic predicate only generates clock-bounded sequences

We need to show that all behavioural sequences generated by *smPred* belong to *ckBnd* subtype. To do so, we first need two lemmas proving that the semantic domain operators preserve the *ckBnd* property. Regarding the application of $^\frown$ to clock-bounded sequences, it is not possible to prove that the concatenation of two *ckBnd* sequences is still a clock-bounded sequence (it is not even possible to apply $^\frown$ as the arguments do not satisfy their precondition). On the other hand, it is possible to prove that:

Lemma 5.2.1.

$$\forall S_1, S_2 \in \mathcal{S}eq, a \in \mathscr{P}(Action) \bullet ckBnd(S_1) \land ckBnd(S_2) \Rightarrow ckBnd(S_1{}^\frown(\mathcal{C}b\ a\ S_2)).$$

This result is strong enough to aid us in the proof of *smPred*'s preservation of the *ckBnd* property. In fact, it is easy to observe that the way in which the concatenation function is applied in the above lemma is the only way in which the function is applied in *smPred*'s definition.

The proof of the lemma stating $\uplus$'s monotonicity with respect to the *ckBnd* property is very complicated. The complication arises because of the way in which the *ckBnd*'s induction principle has to be applied (i.e., in sequential order) together with $\uplus$'s definition. In order to overcome this problem, we capture $\uplus$'s behaviour in the inductive predicate $\uplus_{pred}$. The predicate $\uplus_{pred}(S_1, S_2, S_3)$ relates the pair of sequences S_1 and S_2 with the sequence S_3 provided S_3 can be obtained from merging S_1 with S_2 following the inductive definition below.

Definition 5.2.1.

$$\forall s \bullet \uplus_{pred}(\bot, s, \bot)$$
$$\forall s \bullet \uplus_{pred}(s, \bot, \bot)$$
$$\forall ca_1, ca_2 \bullet \uplus_{pred}((\mathtt{Ck}\ ca_1), (\mathtt{Ck}\ ca_2), (\mathtt{Ck}\ ca_1 \cup ca_2))$$
$$\forall ca_1, ca_2, s, a, s_1 \bullet ckBnd(s) \land (s = (\mathtt{Ck}\ ca_1\ (\mathtt{Cb}\ a\ s_1))) \Rightarrow$$
$$\quad \uplus_{pred}(s, (\mathtt{Ck}\ ca_2), (\mathtt{Ck}\ ca_1 \cup ca_2\ (\mathtt{Cb}\ a\ s_1)))$$
$$\forall ca_1, ca_2, s, a, s_1 \bullet ckBnd(s) \land (s = (\mathtt{Ck}\ ca_2\ (\mathtt{Cb}\ a\ s_1))) \Rightarrow$$
$$\quad \uplus_{pred}((\mathtt{Ck}\ ca_1), s, (\mathtt{Ck}\ ca_1 \cup ca_2\ (\mathtt{Cb}\ a\ s_1)))$$
$$\forall ca_1, ca_2, s, a_1, a_2, s_1, s_2, merge_{s_1,s_2} \bullet \uplus_{pred}(s_1, s_2, merge_{s_1,s_2}) \Rightarrow$$
$$\quad \uplus_{pred}((\mathtt{Ck}\ ca_1\ (\mathtt{Cb}\ a_1\ s_1)), (\mathtt{Ck}\ ca_2\ (\mathtt{Cb}\ a_2\ s_2)), (\mathtt{Ck}\ ca_1 \cup ca_2\ (\mathtt{Cb}\ a_1 \cup a_2\ merge_{s_1,s_2}))).$$

From this definition, it is easy to show the above property for the $\uplus_{pred}$ predicate (i.e., it preserves the *ckBnd* property):

Lemma 5.2.2.

$$\forall S_1, S_2, S_{rest} \in \mathbb{S}eq\bullet \\ ckBnd(S_1) \land ckBnd(S_2) \land \uplus_{pred}(S_1, S_2, S_{rest}) \Rightarrow ckBnd(S_{rest}).$$

The main advantage of the predicate-based form of $\uplus$ is that it provides an induction principle, allowing us to conduct proofs by induction on the merge operator, rather than on sequences or its properties. Before being able to take advantage of this feature we need to prove that it is safe to replace $\uplus$ by the $\uplus_{pred}$ inductive predicate (i.e., we need to prove that they are equivalent). As we intend to replace $\uplus$ by $\uplus_{pred}$ only in the context of our semantic predicate, we can safely assume that the *ckBnd* property holds and prove the equivalence result that we need by showing mutual implication between the two formulations for the parallel-merge operator:

Lemma 5.2.3.

$$\big(\forall S_1, S_2, r \in \mathbb{S}eq \bullet ckBnd(S_1) \land ckBnd(S_2) \land \uplus_{pred}(S_1, S_2, r) \Rightarrow (r = S_1 \uplus S_2)\big) \land \\ \big(\forall S_1, S_2 \in \mathbb{S}eq \bullet ckBnd(S_1) \land ckBnd(S_2) \Rightarrow \uplus_{pred}(S_1, S_2, S_1 \uplus S_2)\big).$$

With the equivalence result above we show that Lemma 5.2.2 also holds for $\uplus$:

Lemma 5.2.4.

$$\forall s_1, s_2, res \in \mathbb{S}eq \bullet ckBnd(s_1) \land ckBnd(s_2) \Rightarrow ckBnd(s_1 \uplus s_2).$$

The two main lemmas of this section allow us to prove that the behavioural sequences generated by *smPred* belong to the subset of Seq induced by *ckBnd*:

Theorem 5.2.1.

$$ckBnd_{seq_c} \vdash \forall c \in Lang; init_c, link_c \in \mathbb{P}(Action); seq_c \in \mathbb{S}eq\bullet \\ smPred\big(c, \big(\big[(init_c\ seq_c\ link_c)\big]\big)\big) \Rightarrow ckBnd(seq_c)\big).$$

This final result ensures that the operators in the semantic domain are always applied within their definition domain by the semantic predicate.

6. Wire-wise correctness

We are now in a good position to verify the correctness of the wiring schema used to link the different components used at the hardware level. In particular, we are interested in proving the wire-correctness of the generated hardware by means of verifying whether: (a) the activation signal is propagated from the *finish* signal of the previous circuit; (b) the start signal is given to each component at the right clock cycle; and (c) the internal wiring of each circuit propagates the control signals in the right way and produces the *finish* pulse at the right time.

It is worthwhile noting that part of the verification is straightforward from the way in which the compilation is done. In particular, each construct is compiled into a *black box* and it is interfaced only through its *start* and *finish* wires. In this sense, it is impossible for a circuit to be started by any component but the circuit containing it, pre-empting the chance of a component *c* being activated by a random piece of hardware. After this observation, to prove (a) we only need to prove that the *finish* wire of the appropriate circuit is set to *high* by the time the subsequent circuit is started.

Regarding the verification of the *start* signal given at the right time (condition b), we have already included assertions regarding the *start* signal in the combinatorial prelude of all constructs in order to make sure that the circuit receives a *start* pulse during the first clock cycle of its execution. The remaining aspect of this question is whether our semantic model of the hardware activates the circuits at the right clock cycle. Towards answering this question, the synchronous time model used in Handel-C together with the component-based approach used in the compilation allows us to verify that the timing in our semantic model is equivalent to the one of the generated hardware. In this context, assuming that the generated hardware implements the timing model correctly, we only need to verify that the wire-related assertions are satisfied in order to verify (b).

The rest of this section is devoted to verifying the wire-correctness of the hardware on the basis of the observations above. We first define a way of calculating whether a given wire is *high* within the current clock cycle. We then define the concept of wire-satisfiability capturing the notion of wire-based assertions being true in a given set of combinatorial actions and use it to prove all the circuits are given the start signal in the clock cycle they are supposed to be started in.

6.1. Wire-transfer closure

The fact that all the combinatorial actions happening at a given clock cycle are collected together in a set provides enough information to allow one to "calculate" which wires hold the *high* value in that clock cycle. This is because of the way in

which clock edges are handled: all the wires are set to *low*, forcing the presence of explicit combinatorial actions to set the appropriate wires to *high* again in the next clock cycle. In fact, the information about a wire holding the *high* value can be given either by a single formula (such as $\pi_s(c)$) or by a chain of value transfers from other wires (such as $\{w_1 \hookleftarrow w_2 ; w_2\}$). In this context, we define the notion of a wire in *high* by means of the *isHigh* predicate:

Definition 6.1.1.

$$\begin{array}{l}
\forall\, w_1, set \bullet (w_1 \in set) \Rightarrow isHigh(w_1, set)) \wedge \\
\forall\, w_1, w_2, w_3, set \bullet isHigh(w_2, set) \wedge isHigh(w_3, set) \wedge \\
\qquad\qquad (w_1 \hookleftarrow (w_2 \wedge w_3) \in set) \Rightarrow isHigh(w_1, set) \wedge \\
\forall\, w_1, w_2, w_3, set \bullet isHigh(w_2, set) \vee isHigh(w_3, set) \wedge \\
\qquad\qquad (w_1 \hookleftarrow (w_2 \vee w_3) \in set) \Rightarrow isHigh(w_1, set) \wedge \\
\forall\, w_1, w_2, set \bullet isHigh(w_2, set) \wedge (w_1 \hookleftarrow w_2 \in set) \Rightarrow isHigh(w_1, set).
\end{array}$$

At this point it might be relevant to highlight the fact that the definition above provides the induction rules that are used to generate the relation *isHigh* (as mentioned by Melham [18], "inductive relations are based on the concept of a relation being closed under a set of rules"). In HOL, an inductively defined definition like the one above adds three elements to a theory: (a) *induction rules*, (the rules above); (b) the *induction principle*, capturing how structural induction is to be conducted on the basis of the induction rules; and (c) a *case analysis theorem*, allowing the derivation of the relation's rules from the conclusions. As an example, we present below the case analysis theorem generated for our *isHigh* predicate above.

Theorem 6.1.1.

$$\begin{array}{l}
\forall a \in (\mathtt{Action} \times \mathscr{P}(\mathtt{Action})) \bullet \\
\quad isHigh(a) \Leftrightarrow \\
\qquad (\exists w_1, set \bullet (a = (w, set)) \wedge w \in set) \vee \\
\qquad (\exists w_1, w_2, w_3, set \bullet (a = isHigh(w_1, set)) \wedge \\
\qquad\qquad isHigh(w_2, set) \wedge isHigh(w_3, set) \wedge (w_1 \hookleftarrow (w_2 \wedge w_3) \in set)) \vee \\
\qquad (\exists w_1, w_2, w_3, set \bullet (a = (isHigh(w_1, set))) \wedge \\
\qquad\qquad (isHigh(w_2, set) \vee isHigh(w_3, set)) \wedge (w_1 \hookleftarrow (w_2 \vee w_3) \in set)) \vee \\
\qquad (\exists w_1, w_2, set \bullet (a = (isHigh(w_1, set))) \wedge isHigh(w_2, set) \wedge (w_1 \hookleftarrow w_2 \in set)).
\end{array}$$

The predicate *isHigh* captures the notion of a wire w holding the high value provided the actions in *set* are executed. From this definition we were able to prove some lemmas that will be necessary in the following sections. In particular, if a given wire w holds the high value in a given set s, then it still does so in a bigger set:

Lemma 6.1.1. $isHigh(w, s) \Rightarrow isHigh(w, (s \cup s_1))$.

It is also possible to prove that any explicit action in the set of actions setting up a wire w_1 to the *high* value can be replaced by a pair of actions, one setting up a new wire w_2 to the *high* value and another one to propagate its value into w_1:

Lemma 6.1.2.

$$isHigh(w, s \cup \{w_1\}) \Rightarrow isHigh(w, s \cup \{w_2 ; w_1 \hookleftarrow w_2\}).$$

With these results we are able to prove that for any given construct c the epilogue in the semantics always sets its finish wire $\pi_f(c)$ to *high*:

Theorem 6.1.2.

$$\begin{array}{l}
smPred\big(c, \big[\, init_c\ seq_c\ link_c \,\big]\big) \wedge (\pi_s(c) \in init_c) \Rightarrow isHigh(\pi_f(c), link_c) \wedge \\
smPred(c, [\![init_c\ link_c]\!]) \wedge (\pi_s(c) \in init_c) \Rightarrow isHigh(\pi_f(c), link_c).
\end{array}$$

This is the first result towards proving (a) in the introduction of this section. In order to complete our verification of (a) we introduce a new assertion type $(w_1 \rightsquigarrow w_2)_\perp$ defined to hold iff $isHigh(w_1) \wedge isHigh(w_2)$ holds in a given set of combinatorial actions. We then modify our semantic predicate to include assertions of this new type at the places where condition (a) is expected to hold. As an example, we show the updated version of the semantics for the sequential composition construct:

Definition 6.1.2.

$$\begin{array}{l}
\forall\, s, f, c_1, init_{c_1}, seq_{c_1}, link_{c_1}, c_2, init_{c_2}, seq_{c_2}, link_{c_2} \bullet \\
\quad smPred\big(c_1, \big[init_{c_1}\ seq_{c_1}\ link_{c_1}\big]\big) \wedge smPred\big(c_2, \big[init_{c_2}\ seq_{c_2}\ link_{c_2}\big]\big) \Rightarrow \\
\qquad smPred\big((\mathtt{Scomp}\ s\ f\ c_1\ c_2), \big[init_{c_1} \cup \{s_\perp ; \pi_s(c_1) \hookleftarrow s\} \\
\qquad\qquad\qquad seq_{c_1} \frown (\mathtt{Cb}\ (link_{c_1} \cup init_{c_2} \cup \{\pi_s(c_2) \hookleftarrow \pi_f(c_1)\} \cup \\
\qquad\qquad\qquad\quad \{(\pi_f(c_1) \rightsquigarrow \pi_s(c_2))_\perp\})\ seq_{c_2}) \\
\qquad\qquad\qquad link_{c_2} \cup \{f \hookleftarrow \pi_f(c_2)\}\big]\big).
\end{array}$$

In order to complete the verification, we need a way to ensure that the new assertion holds for any possible semantic trace generated from the compilation. The next subsection provides the mechanisms needed to prove that this is the case.

6.2. Assertion satisfiability

Having defined the concept of wire being set to *high*, we need a way to capture the idea of satisfaction of our assertions regarding wires. In particular, we say that all the wire-related assertions in a set are *satisfied* iff the predicate *wireSAT* holds[3]:

Definition 6.2.1.

$$wireSAT(s) = \forall\, w \in WireIds \bullet w_\perp \in s \Rightarrow isHigh(w_\perp, s).$$

The hardware generated from any given syntactic construct c is started; then all the wire-related assertions in its combinatorial prelude hold:

Theorem 6.2.1.

$$smPred\big(c, \big[init_c\ seq_c\ link_c\big]\big) \Rightarrow wireSAT(init_c \cup \{\pi_s(c)\}) \wedge$$
$$smPred(c, [\![init_c\ link_c]\!]) \Rightarrow wireSAT(init_c \cup \{\pi_s(c)\}).$$

This result is proving consideration (a) from the previous section: the activation signal is propagated from the parent/previous circuit into the start signal of the current one. In fact, even though the above theorem is just stating that it happens that the right *start*/*finish* signals get the high value in the appropriate clock cycle, evidence gathered during the proof process showed that the *high* value actually gets propagated between them, proving consideration (a) to its full extent. It also shows that consideration (b) holds: the start *signal* is given to each circuit at the appropriate time (provided that the time models of the hardware compilation are correct as regards Handel-C's semantics).

We also proved that the epilogue set of combinatorial actions satisfies *wireSAT*:

Lemma 6.2.1.

$$smPred\big(c, \big[init_c\ seq_c\ link_c\big]\big) \Rightarrow wireSAT(link_c) \wedge$$
$$smPred(c, [\![init_c\ link_c]\!]) \Rightarrow wireSAT(link_c).$$

Provided that (a) and (b) hold, the verification of (c) can be reduced to proving that the assertions in the behavioural part of the semantic predicate are satisfied. The rationale behind this affirmation is that the base cases for the *smPred* trivially satisfy (c) while compound circuits can be regarded as placeholders linking *start*/*finish* wires of different components by means of combinatorial actions. The assertions introduced in order to verify (a) and (b) are checking that those actions are propagating the right value among the different components involved.

In order to verify the behavioural sequences from the semantic predicate we first need to extend the concept of wire-satisfiability to sequences. We do so by defining the function $wireSAT_{seq}$ as follows:

Definition 6.2.2.

$$wireSAT_{seq}(\perp) = F \wedge wireSAT_{seq}(\langle\rangle) = T$$
$$wireSAT_{seq}(\mathtt{Cb}\ a\ s_1) = wireSAT(a) \wedge wireSAT_{seq}(s_1)$$
$$wireSAT_{seq}(\mathtt{Ck}\ a\ s_1) = wireSAT_{seq}(s_1).$$

Before being able to use the $wireSAT_{seq}$ function to prove that the control flow wiring is correct in the behavioural sequences we need to prove two lemmas as regarding $^\frown$ and $\uplus$ preserving the wire-satisfiability property for sequences if it holds true for their arguments. The case of concatenation is straightforward by induction over the composed sequences:

Lemma 6.2.2.

$$wireSAT_{seq}(s_1 {}^\frown s_2) \Leftrightarrow wireSAT_{seq}(s_1) \wedge wireSAT_{seq}(s_2).$$

Proving the equivalent result for the parallel-merge operator is a very complicated task given the already mentioned lack of an induction principle capable of handling two sequences as a pair. As we are still within the context of the semantic predicate (and hence, within the subclass of clock-bounded sequences) we can prove an easier goal:

Lemma 6.2.3.

$$wireSAT_{seq}(s_1) \wedge wireSAT_{seq}(s_2) \wedge \uplus_{pred}(s_1, s_2, res) \Rightarrow wireSAT_{seq}(res)$$

and then use the equivalence between *parMerge* and $\uplus$ to deduce the equivalent result for $\uplus$. With these two results, we are able to prove the correctness of the wiring for the behavioural sequences produced by *smPred*:

Theorem 6.2.2.

$$smPred(c, \big[init_c\ seq_c\ link_c\big]) \Rightarrow wireSAT_{seq}(seq_c).$$

With the three main theorems of this section, we show that the wiring is correct (regarding our wire-satisfiability criteria) for any given construct in the core language.

[3] We replace assertions of the form $(w_1 \leftrightsquigarrow w_2)_\perp$ by the equivalent set of assertions $\{(w_1)_\perp\ ;\ (w_2)_\perp\}$, allowing us to use the simple form of satisfiability defined before.

7. Conclusions and future work

The main contributions of this work are a compilation schema for the `priAlt` construct, an improved semantic model for the hardware components synthesised from Handel-C programs and the mechanical verification of the wiring schema used to handle the control flow among those components.

This work presents a more abstract semantic domain than the one used in previous works and allows a better description of the parallelism exhibited by the synthesised hardware. In particular, we have defined our semantic domain in terms of a deep embedding of sequences of state-transformers in higher order logic. We have also established a partial order relationship over the domain and proved the existence of fix-point solutions to our inductive approximations for recursive constructs.

The synthesis process we are formalising is based on the assumption that no hardware component will be activated unless a precise signal has been given to it through its interface. We have captured this notion by embedding Handel-C's syntactic constructs in HOL and providing a semantic function that maps each construct in the language to its representation in our semantic model. Moreover, the way in which Handel-C's synchronous nature is encoded in the model introduces explicit information about the value held by the wires used to link different components. We have taken advantage of this feature to formally verify the correctness of the wiring schema used in the compilation.

Even though we have proved that each of the hardware components propagates the control token in the right way, we still need to prove that the hardware generated by the compilation rules is *correct* (i.e., semantically equivalent to its original Handel-C code). This correctness proof will also allow us to discharge the only assumption of this work: that the timing model of the generated hardware is consistent with the one for Handel-C. Towards this end, our next step is to prove the existence of an equivalence relationship using the semantic models for Handel-C [5,23] and the semantics for the hardware generated that is presented in this paper.

Appendix. Supplementary data

Supplementary data associated with this article can be found, in the online version at doi:10.1016/j.scico.2010.02.007.

References

[1] R. Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, John Van Tassel, Experience with embedding hardware description languages in HOL, in: Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, 1992, pp. 129–156.
[2] S. Brookes, On the axiomatic treatment of concurrency, in: Seminar on Concurrency [3], pp. 1–34.
[3] Stephen D. Brookes, A.W. Roscoe, Glynn Winskel (Eds.), Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9–11, 1984, in: Lecture Notes in Computer Science, vol. 197, Springer, 1985.
[4] A. Butterfield, Denotational semantics for prialt-free Handel-C. Technical report, The University of Dublin, Trinity College, December 2001.
[5] A. Butterfield, A denotational semantics for Handel-C, in: Formal Methods and Hybrid Real-Time Systems, 2007, pp. 45–66.
[6] J. Camilleri, T. Melham, Reasoning with Inductively Defined Relations in the HOL Theorem Prover, Technical Report 265, August 1992.
[7] B. Davey, H. Priestley, Introduction to Lattices and Order, Cambridge University Press, 2002.
[8] R. Floyd, Assigning meaning to programs, in: Mathematical Aspects of Computer Science, vol. 19, American Mathematical Society, 1967, pp. 19–32.
[9] M. Gordon, T. Melham (Eds.), Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, Cambridge University Press, 1993.
[10] C.A.R. Hoare, Communicating sequential processes, Communications of the ACM 26 (1) (1983) 100–106.
[11] C.A.R. Hoare, J. He, Unifying Theories of Programming, Prentice Hall, 1998.
[12] Brian W. Kernighan, The C Programming Language, Prentice Hall Professional Technical Reference, 1988.
[13] Christian Lengauer, Chua-Huang Huang, The static derivation of concurrency and its mechanized certification, in: Brookes et al. [3], pp. 131–150.
[14] Celoxica Ltd., DK3: Handel-C Language Reference Manual, 2002.
[15] Celoxica Ltd., The Technology behind DK1, August 2002. Application Note AN 18.
[16] T. Melham, Using recursive types to reason about hardware in higher order logic, in: The Fusion of Hardware Design and Verification, 1988, pp. 27–50.
[17] T. Melham, Higher Order Logic and Hardware Verification, in: Cambridge Tracts in Theoretical Computer Science, vol. 31, Cambridge University Press, 1993.
[18] T.F. Melham, A package for inductive relation definitions in HOL, in: Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, IEEE Computer Society Press, 1992, pp. 350–357.
[19] I. Page, Constructing hardware–software systems from a single description, Journal of VLSI Signal Processing 12 (1) (1996) 87–107.
[20] I. Page, W. Luk, Compiling Occam into field-programmable gate arrays, in: Oxford Workshop on Field Programmable Logic and Applications, 1991, pp. 271–283.
[21] J. Perna, J. Woodcock, A denotational semantics for Handel-C hardware compilation, in: Michael Butler, Michael G. Hinchey, María M. Larrondo-Petrie (Eds.), ICFEM, in: Lecture Notes in Computer Science, vol. 4789, Springer, 2007, pp. 266–285.
[22] J. Perna, J. Woodcock, Proving wire-wise correctness for Handel-C compilation in HOL, Technical Report YCS-2008-429, Computer Science Department, The University of York, December 2007.
[23] J. Perna, J. Woodcock, UTP semantics for Handel-C, in: Unifying Theories of Programming, Second International Symposium, UTP2008, in: Lecture Notes in Computer Science, vol. 5713, Springer, 2008, pp. 142–161.
[24] J. Perna, J. Woodcock, Mechanised wire-wise verification of Handel-C Synthesis, Electronic Notes in Theoretical Computer Science 240 (2009) 201–219.
[25] Rachel E.O. Roxas, A HOL package for reasoning about relations defined by mutual induction, in: HUG'93: Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications, Springer-Verlag, London, UK, 1994, pp. 129–140.
[26] D. Scott, C. Strachey, Towards a mathematical semantics for computer languages, in: J. Fox (Ed.), Proceedings Symposium on Computers and Automata, Pol. Inst. of Brooklyn Press, New York, 1971, pp. 19–46.